Using MMX™ Technology Instructions to Implement a 2/3T Equalizer

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

CONTENTS

1.0. INTRODUCTION

The Intel Architecture media extensions include single-instruction, multi-data (SIMD) instructions. This application note presents example code that implements a 2/3T Spaced Equalizer algorithm on complex arithmetic data using the media extensions. First, a brief description of the 2/3T Spaced Equalizer algorithm is given below. Next, the C language implementation of the 2/3T Spaced Equalizer algorithm is analyzed and partitioned from a performance efficiency viewpoint. Subsequently, its implementation and optimization using MMX technology instructions are discussed. The improved performance is achieved by using instruction pairing and MMX technology features such as data packing, performing four word multiplies and two double word adds in three cycles.

2.0. THE 2/3T SPACED EQUALIZER ALGORITHM

This algorithm is an example of an adaptive filter. It primarily consists of three loops, which perform the following operations (refer to the C code given in Appendix A for details):

```
Loop A: ( Loop over input sequence length )
                Tempreal[k]=Inputreal[i];
                Tempimag[k]= Inputimag[i];
                where k = (a constant) * 2 + i;  i = 0, 1, ..N-1
Loop B: ( Loop over one-third the input sequence length )
```

This loop consists of two inner loops with a scalar code segment between these two inner loops. The primary functionality performed during each pass of this loop can be categorized into the following three basic operations:

Step 1. Filters the input data (loop over filter coefficient length).

This operation involves a complex multiplication (four multiply and two add operations), and a complex summation (two add operations).

```
Sumreal = (Inputreal(k) * Coeffreal(i) - Inputimag(k) * Coeffimag(i))
Sumimag = (Inputimag(k) * Coeffreal(i) + Inputreal(k) * Coeffimag(i))
where i = 0, 1, 2.......N-1 and k = (a constant) + 2 * i
N denotes the number of filter coefficients.
```

Step 2. Stores the output and estimates the error (scalar code segment).

This involves conditional assignments, four add and four shift operations. This is a scalar segment of code and no loops are involved.

```
Outputreal[k] = (Sumreal+0x4000)>>14;
Outputimag[k] = (Sumimag+0x4000)>>14;
if (Outputreal[k] >= 0)
    Errorreal =  (constanta - outputreal[k])>>4 ;
else
    Errorimag =  (constantb - outputimag[k])>>4 ;
```

Step 3. Adapts/updates the filter coefficients based on the estimated error (loop over filter coefficient length).

This requires a complex multiplication (four multiply and two add operations), four addition and two shift operations. The result is finally saturated to a 16-bit fixed-point representation.

```
        Temp = Errorreal * Inputreal(k) + Errorimag * Inputimag(k)
        Coeffreal(i) = [(Temp + 0x4000) >> 15] + Coeffreal(i)
        Temp = Errorimag * Inputreal(k) - Errorreal * Inputimag(k)
        Coeffimag(i) = [(Temp + 0x4000) >> 15] + Coeffimag(i)
        where i = 0, 1, 2...N-1 and k = (a constant) + 2*i
        denotes the number of filter coefficients.
Loop C: (Loop over twice the filter coefficient length)
        Tempreal[i]= Tempreal[k];
        Tempimag[i]= Tempimag[k];
        where k = (a constant) + i;  i = 0, 1, 2..N-1
```

are represented as complex 16 bit fixed point quantities.

3.0. CODE PARTITIONING AND DATA STORAGE FORMAT

In order to create an MMX technology implementation of the compute-intensive segments in the code, it was decided to retain loops A and C (mentioned above) as part of the C code and implement only loop B as part of the MMX technology code (see Appendix B).

Note that in loop B, to perform the filtering operation (step 2), we require the real and imaginary components of the filter coefficients (*hI* and *hQ*, respectively) as well as of the input data (*sI* and *sQ*). Since all these are word (16-bit) quantities, we could exploit the PMADDWD instruction to perform the entire complex multiplication in three cycles. This can be accomplished only if the data quantities are stored in the right format and can be fetched as quadwords (64-bit). The subtraction operation involved in the computation of the real component requires that data be not only properly formatted, but also properly modified. The two options available for storing these information (as quadwords) to perform the computation of step 1 are:\\

63		31	0		63		31	0
hI	hQ	hI	hQ	OR	hI	hQ	hQ	hI
sI	sQ	sQ	sI		sI	sQ	sI	sQ

In the option on the left, we need a negative *sQ*, In the option on the right, we need a negative *hQ*. Note that if we chose to implement the second compute option (the one where negative *hQ* is stored, during adaptation (step 3), it would be difficult to update and store the updated values of *hI* and *hQ* in the same format (*hI, -hQ, hQ, hI*). On the other hand, *sI* and *sQ* are not modified by the code. Hence, it would be easier to store (*sI, sQ*) in the modified *form (sI,-sQ,sQ,sI*). Consequently, we chose the first compute option for representing (*hI, hQ*) and (*sI, sQ*). This also facilitates simultaneous execution of step 1 for the real and imaginary components of the output. After PMADDWD, the upper 32 bits correspond to the real part and the lower 32 bits correspond to the imaginary part. Of course, the (16 bit) real and imaginary components of the output (*yI* and *yQ*) can be packed as one DWORD (32 bit) quantity and stored with one memory store operation.

Note that during adaptation the error term (*eI* and *eQ*) is multiplied with the complex conjugate of *sI* and *sQ*. However, to facilitate the filtering operation, *sI* and *sQ* were stored (discussion above) in a format NOT conducive to this multiplication. This complex multiply can still be performed efficiently by properly formatting the error terms. Note that after the execution of the filter operation, the real and imaginary parts of the output are stored in the lower 32 bits of a register. Straightforward processing of this results in the real and imaginary components of the error term also being stored in the same format. This will not facilitate the performance of the complex multiply operation in the adaptation loop because of the format in which (*sI, sQ*) are stored. Hence, the error term will have to be formatted accordingly. This is *not* an expensive formatting operation since step 2 is a scalar segment within loop B (unlike steps 1 and 3 which are loops themselves). From the required complex multiplication and the above-mentioned storage scheme for (*sI, sQ*), we arrive at the following format for the error term:

	eI	eQ	eI	eQ	
and	sI	sQ	sQ	sI	(as stored above).

With these data formats, the complex multiply operations in both steps 1 and 3 can be performed with one PMADDWD instruction (executes in three cycles).

To implement these, the assignment operations in loops A and C should also be properly modified in the C code. The modified C code is shown in Appendix B.

Observe that by storing (*sI, sQ*) as a 8 byte quantity, the (*sI, sQ*) index computation in steps 1 and 3 (within loop B) will always result in quad-aligned memory accesses. Likewise, (*hI, hQ*) and (*yI, yQ*) fetches and stores, respectively, always result in DWORD-aligned memory accesses.

Penalty:

While all of these data formatting steps have enhanced the performance, the memory requirement for this algorithm has increased. The (*sI, sQ*) data which originally required four bytes of storage, now needs eight bytes. This, in most cases, is not a serious penalty. The memory requirement for (*hI, hQ*), however, need not be doubled since it can be easily formatted into the desired form by using the UNPACK instructions available as part of the MMX instruction set:

Final data formats in memory:

63			0		31	0		31	0
sI	sQ	sQ	sI		hI	Hq		yI	yQ

where (*sI, sQ*) are the input data terms, (*hI, hQ*) are the filter coefficient terms and (*yI, yQ*) are the output terms.

4.0. THE FILTER OPERATION (STEP 1 OF LOOP B)

The basic filter operation code is given below. The (*hI, hQ*) values are stored in memory pointed to by register ebx while the (*sI, sQ*) values are stored in memory pointed to by register EAX. The partial sum is computed and stored in register MM7. The (*hI, hQ*) values are unpacked in instruction 2 to prepare for the complex multiplication performed in instruction 3. Please refer to the assembly code in Appendix C during the discussion in this section. Of these four instructions, only instruction 3 is a multi-cycle instruction (three cycles). So each iteration through this loop takes six cycles. Also, none of these instructions can be paired for execution in the same cycle (due to register data dependencies). Hence, for a fully optimized implementation, we have at our disposal, four half cycles that might pair with each of these instructions and two idle cycles between instructions 3 and 4. These provide for a potential of eight additional instructions.

Table 1 . Basic Filter Loop in MMX™ Technology

Instruction Number	Command	Operands
1	MOVD	MM1, [EBX]
2	PUNPCKLDQ	MM1, MM1
3	PMADDWD	MM1, [EAX]
4	PADDD	MM7, MM1

In order to take advantage of these idle cycles, we could fetch and process the (*hI, hQ*) and (*sI, sQ*) elements for the next iteration of this loop. Given that each iteration requires four instructions and there is a potential for a total of eight additional instructions, we could process the next two iterations also within this loop. By doing so, we arrive at the resulting code for the optimized loop.

Table 2. Optimized Filter Loop

Clock Number	Pipe	Data Set	Command	Operands
1	U Pipe	first	MOVD	MM1, [EBX]
1	V Pipe	second	PADDD	MM7, MM3
2	U Pipe	second	MOVD	MM3, 4[EBX]
2	V Pipe	first	PUNPCKLDQ	MM1, MM1
3	U Pipe	first	PMADDWD	MM1, [EAX]
3	V Pipe	third	PADDD	MM7, MM5
4	U Pipe	third	MOVD	MM5, 8[EBX]
4	V Pipe	second	PUNPCKLDQ	MM3, MM3
5	U Pipe	second	PMADDWD	MM3, 16[EAX]
5	V Pipe	third	PUNPCKLDQ	MM5, MM5
6	U Pipe	third	PMADDWD	MM5, 32[EAX]
6	V Pipe	first	PADDD	MM7, MM1

While ordering these instructions within the loop, attention was paid to the number of cycles each instruction would take before its results would be available for further computation. Another important aspect considered was to ensure that each instruction could be issued to its corresponding U or V pipe in the processor. For example, care was taken to ensure that the memory fetch/store operations would be

issued only to the U pipe. Note that if the length of this loop were not a multiple of three, then one or two sets of elements would have to be computed outside of the loop to take the proper boundary condition into account.

Note that in this case, we did not add any new cycles while attempting to process three elements within each loop. This results in a performance of two cycles per iteration as opposed to the initial implementation which has a performance of six cycles per iteration. This may not always be possible. Based on each application, one will have to evaluate the cycles per iteration. In some cases, it is possible that this number might improve despite the addition of a few cycles to the loop.

5.0. ADAPTATION/UPDATE OF FILTER COEFFICIENTS (STEP 3 OF LOOP B)

The basic MMX technology version of the adaptation/update operation is given in the table below. Once again, the ebx register points to the memory location where (*hI, hQ*) are stored and likewise the eax register points to the memory location for (*sI, sQ*). The complex error term computed earlier is stored in register mm4 while register mm5 contains the rounding constant. Please refer to the assembly code in Appendix C during the discussion in this section.

Instructions 1 and 2 could have been combined into one instruction (PMADDWD MM4, [EAX]). This requires that MM4 be the destination register. But this destroys the error term stored earlier in register MM4. Hence, to retain the error term in register mm4, the memory operand is copied to register mm0 in instruction 1.

Instructions 5, 6 and 7 are used to format the partial result in the desired format so that the complex saturated addition with (*hI, hQ*) can be executed in one cycle. Refer above to the discussion on the data storage format to observe that the error term is stored in a format different from the format in which (*hI, hQ*) are stored. These three instructions are the penalty one pays for choosing the above-mentioned storage format. Once again due to data dependencies, most of these instructions cannot be paired. Here again, PMADDWD is the only multi-cycle (three cycles) instruction. Also, like in the filter loop, not much pairing related optimization can be achieved within each iteration due to data dependencies with the loop as given in the following table.

Table 3. Basic Filter Adaptation Loop in MMX™ Technology

Instruction Number	Command	Operands
1	MOVQ	MM0, [EAX]
2	PMADDWD	MM0, MM4
3	PADDD	MM0, MM5
4	PSRAD	MM0, 15
5	MOVQ	MM7, MM0
6	PUNPCKHDQ	MM7, MM7
7	PUNPCKLWD	MM0, MM7
8	PADDSW	MM0, [EBX]
9	MOVD	[EBX], MM0

As in the filter loop**,** doing a second set of elements within the same iteration of this loop allows much better pairing. It also allows useful work while waiting for the multiplier results. This results in the following table:

Table 4. Optimized Filter Adaptation Loop

Instruction Number	Pipe	Data Set	Command	Operands
1	U Pipe	First	MOVQ	MM0, [EAX]
1	V Pipe	Second	PADDD	MM2, MM5
2	U Pipe	Second	MOVD	MM3, 4[EBX]

2	V Pipe	First	PMADDWD	MM0, MM4
3	U Pipe	First	MOVD	MM1, [EBX]
3	V Pipe	Second	PSRAD	MM2, 15
4	U Pipe	Second	MOVQ	MM6, MM2
4	V Pipe	Second	ADD	EAX, 16
5	U Pipe	First	PADDD	MM0, MM5
5	V Pipe	Second	PUNPCKHDQ	MM6, MM6
6	U Pipe	First	PSRAD	MM0, 15
6	V Pipe			
7	U Pipe	First	MOVQ	MM7, MM0
7	V Pipe	Second	PUNPCKLWD	MM2, MM6
8	U Pipe	First	PUNPCKHDQ	MM7, MM7
8	V Pipe	Second	PADDSW	MM3, MM2
9	U Pipe	Second	MOVD	4[EBX], MM3
9	V Pipe	First	PUNPCKLWD	MM0, MM7
10	U Pipe	Second	MOVQ	MM2, 32[EAX]
10	V Pipe	First	PADDSW	MM1, MM0
11	U Pipe	First	MOVD	[EBX], MM1
11	V Pipe	Second	PMADDWD	MM2, MM4

Since the number of instructions required to execute in each iteration was greater than the number of idle cycles available in the basic implementation, computation of the second set of elements was initiated prior to entering this loop. This helped ensure optimized execution of the loop with all the instructions being paired except one. Also, by updating the eax register in two steps instead of one, data integrity was ensured during the computation of the two sets of elements within each iteration of the loop. Note that if the loop count were not even, then one set of elements would have to be computed outside of the loop to take the proper boundary condition into account.

Several interesting aspects show up in this table. Note that instruction 6 is not paired. Also, relative to the basic implementation, it seems as if a few more cycles have been added. However, notice that the number of cycles per iteration have been reduced. The optimized version computes one iteration in about 5.5 cycles, whereas the scalar version yields about nine cycles per iteration.

6.0. SCALAR CODE SEGMENT (STEP 2 OF LOOP B)

The scalar section of code between the filter loop and the adaptation loop primarily computes the error term. Relative to the two loops, execution of this code segment has a lower impact on the execution speed performance. The principal code optimization techniques employed here are the elimination of conditional jumps and the re-ordering of instructions to give better pairing of instructions.

The if-then-else statement encountered in step 2 of loop B can be implemented without branches by the following sequence of instructions:

Table 5. Branchless Execution of Step 2 in Loop B

Instruction Number	Command	Operands	Comments
1	PCMPEQW	MM0, MM1	compare if equal to zero
2	PCMPGTW	MM2, MM1	compare if greater than zero
3	POR	MM0, MM2	generate boolean pattern for >= zero
4	PAND	MM3, MM0	place constanta if >= zero
5	PANDN	MM0, MM4	place constantb if < zero
6	POR	MM3, MM0	place constant a or constant b depending on >= zero or < than zero
7	PSUBW	MM3, MM1	subtract the output term from the corresponding constant term. This result can then be right shifted by four to give the desired error terms.

In the above table, register mm1 contains outputreal in bits 31 to 16 and outputimag in bits 15 to 0. Registers MM0 and MM2 have been initialized to zeros while registers mm3 and mm4 contain constanta and constantb, respectively. The 16 bit quantities, constanta and constantb, will have to be copied onto bits 31 to 16 as well so that they can be used to operate on both outputreal and outputimag terms. This technique allows the real and imaginary parts to be done in parallel, and also eliminates any delays due to branch mispredictions in the traditional implementation of an if-then-else statement.

APPENDIX A

```
/* The following is the listing of the C code that implements the Equalizer 2/3T
algorithm. */
void Equalizer23 (short *sI,short *sQ,short *hI,short *hQ, short *xI,short *xQ,short
*yI, short *yQ, short h_Leng, short x_Leng)
{
        Complex v, e;
        short i, j;
        long SumI, SumQ;

        for (i = 0; i < x_Leng; i++)     {
                sI[h_Leng*2+i] = xI[i];
                sQ[h_Leng*2+i] = xQ[i];
        }

        for (j = 0; j < x_Leng; j = j+3)  {
        for (SumI = 0, SumQ = 0, I = 0; I < h_Leng; i++) {
             SumI = SumI+sI[3+j+2*i]*(long)hI[i]-sQ[3+j+2*i]*(long)hQ[i];
             SumQ = SumQ+sI[3+j+2*i]*(long)hQ[i]+sQ[3+j+2*i]*(long)hI[i];
        }
           yI[j/3] = (SumI+0x4000)>>14;          // with gain 2
           yQ[j/3] = (SumQ+0x4000)>>14;          // with gain 2
        // generate error
        if (yI[j/3] >= 0)  {v.I = 0x0800;}
        else {v.I = -0x0800;}
        if (yQ[j/3] >= 0)  {v.Q = 0x0800;}
        else {v.Q = -0x0800;}
        e.I = (v.I - yI[j/3])>>4;
        e.Q = (v.Q - yQ[j/3])>>4;

        // adaption
        for (i = 0; i < h_Leng; i++){
        SumI = e.I*(long)sI[3+j+2*i]+e.Q*(long)sQ[3+j+2*i];
        SumQ = e.Q*(long)sI[3+j+2*i]-e.I*(long)sQ[3+j+2*i];
        SumI = ((SumI+0x4000)>>15)+hI[i];
        SumQ = ((SumQ+0x4000)>>15)+hQ[i];
        hI[i] = SumI;
        hQ[i] = SumQ;
        if (SumI > 0x7fff)     { hI[i] = 0x7fff; }
        if (SumI < -0x8000)    { hI[i] = -0x8000; }
        if (SumQ > 0x7fff)     { hQ[i] = 0x7fff; }
        if (SumQ < -0x8000)    { hQ[i] = -0x8000; }
        }
        }
        // update initial state
        for (i = 0; i < 2*h_Leng; i++)
        {
        sI[i] = sI[x_Leng+i];
        sQ[i] = sQ[x_Leng+i];
        }
}
```

APPENDIX B

```
/*
The following is the listing of the modified C code. It is functionally equivalent
to the C code given in Appendix A. This version facilitates optimization in the MMX
technology based implementation of this function.
*/
#include <stdio.h>
extern void FilAdapAsm(short *sP, short *hP, short *yP, short h_Leng, short x_Leng) ;
/*
This routine represents the C code of the function that is implemented/optimized in
assembly(using MMX technology instructions).
*/
void FilAdapC(short *sP, short *hP, short *yP, short h_Leng, short x_Leng) ;void
*AlignAlloc(unsigned int nbytes) ;
void Equalizer23MMx(short *sP,short *hI,short *hQ, short *xI,short *xQ,short *yI,
short *yQ, short h_Leng, short x_Leng)
{
        short i, k ;
        short *hP, *yP ;
        for (i=0; i < x_Leng; i++)  {
        k = (h_Leng*2+i)*4 ;
        sP[k] = xI[i] ;
        sP[k+1] = xQ[i] ;
        sP[k+2] = -xQ[i] ;
        sP[k+3] = xI[i] ;
        /*      sI[h_Leng*2+i] = xI[i];        */
        /*      sQ[h_Leng*2+i] = xQ[i];        */
        }
        /* Allocate the space for interleaving hI and hQ. */
        /* The hI/hQ pair will be orgainzed as: Real Imag. */
        /* Unlike the sI/sQ, the hI/hQ pair will not be duplicated. */
        hP = (short *)AlignAlloc(h_Leng*4) ;
        for(i=0; i < h_Leng; i++) {
                 hP[2*i] = hQ[i] ;
                 hP[2*i+1] = hI[i] ;
        }
        /* Allocate the space for interleaving yI and yQ. */
        /* The yI and yQ output by FilAdapC will be organized as: Real Imag. */
        yP = (short *)AlignAlloc(x_Leng*4) ;
         FilAdapAsm(sP, hP, yP, h_Leng, x_Leng) ;
/*       FilAdapC(sP, hP, yP, h_Leng, x_Leng) ;  */

        /* update initial state  */
        for (i=0; i<2*h_Leng; i++) {
                k = (x_Leng+i)*4 ;
                sP[4*i] = sP[k]  ;
                sP[4*i+1] = sP[k+1] ;
                sP[4*i+2] = sP[k+2] ;
                sP[4*i+3] = sP[k+3] ;
        /*      sI[i]=sI[x_Leng+I];  */
        /*      sQ[i]=sQ[x_Leng+i];  */
        }
        /* Update yI and yQ as modified by FilAdapC */
        for(i=0; i < x_Leng; i=i+3) {
            yI[i/3] = yP[((i/3)*2)+1] ;
```

```
            yQ[i/3] = yP[(i/3)*2] ;
        }
        /* Update hI and hQ as modified by FilAdapC */
        for(i=0; i < h_Leng; i++) {
            hQ[i] = hP[2*i] ;
            hI[i] = hP[2*i+1] ;
        }
}
void FilAdapC(short *sP, short *hP, short *yP, short h_Leng, short x_Leng)
{
        short vI, vQ, eI, eQ;
        short i, j, k;
        long SumI, SumQ;
        for (j=0; j < x_Leng; j=j+3) {
        for (SumI=0, SumQ=0, i=0; i<h_Leng; i++) {
             k = 3+j+2*i ;
             SumI=SumI+sP[4*k]*(long)hP[2*i+1]-sP[4*k+1]*(long)hP[2*i];
             SumQ=SumQ+sP[4*k]*(long)hP[2*i]+sP[4*k+1]*(long)hP[2*i+1];
        }
            k = (j/3)*2 ;
            yP[k+1]=(SumI+0x4000)>>14;   /* with gain 2      */
            yP[k]=(SumQ+0x4000)>>14;     /* with gain 2      */
            /* generate error  */
            if (yP[k+1]>=0) {vI=0x0800;}
            else {vI=-0x0800;}
            if (yP[k]>=0) {vQ=0x0800;}
            else {vQ=-0x0800;}
            eI=(vI-yP[k+1])>>4;
            eQ=(vQ-yP[k])>>4;
            /* adaption    */
            for (i=0; i<h_Leng; i++) {
            k = 3+j+2*i ;
            SumI=eI*(long)sP[4*k]+eQ*(long)sP[4*k+1];
            SumQ=eQ*(long)sP[4*k]-eI*(long)sP[4*k+1];
            SumI = ((SumI+0x4000)>>15)+hP[2*i+1];
            SumQ = ((SumQ+0x4000)>>15)+hP[2*i];
            hP[2*i+1] = SumI ;
            hP[2*i] = SumQ ;
            if (SumI>0x7fff)    { hP[2*i+1]=0x7fff; }
           if (SumI<-0x8000)    { hP[2*i+1]=-0x8000; }
            if (SumQ>0x7fff)    { hP[2*i]=0x7fff; }
            if (SumQ<-0x8000)   { hP[2*i]=-0x8000; }
          }
      }
 }
void *AlignAlloc(unsigned int nbytes)
{
    char *cptr;
    cptr = (char *)malloc(nbytes+8);
    if (!cptr)
    {
        perror("xalloc: Error allocating memory");
        exit(1);
    }
    cptr = (char *)((unsigned long)cptr & 0xFFFFFFF8);
    return(cptr);
}
```

APPENDIX C

```
; The following is the listing of the optimized (using MMX technology ;
  instructions) assembly code.
INCLUDE iammx.inc
    TITLE fil2t3
    486P
; INCLUDE listing.inc
. model FLAT
_DATA SEGMENT
; Define all the constants/local variables
; const_rnd:   to store the rounding value
; pos_bias:    to store the positive bias for clipping
; neg_bias:    to store the negative bias for clipping
; err_fmt_one: pattern "ffff0000 0000ffff" used while
;       formatting the error value
; err_fmt_two: pattern "0000ffff ffff0000" used while
;       formatting the error value
const_rnd              DWORD          4000H, 4000H
pos_                   DWORD          08000800H
neg_bias               DWORD          0f800f800H
err_fmt_one    DWORD          0000ffffH, 0ffff0000H
err_fmt_two    DWORD          0ffff0000H, 0000ffffH
_DATA ENDS
; ****************** ASSUMPTIONS *****************
; The real(sI) & imag.(sQ) terms of the data input are stored as:
;       bits->  63...    ...0
;               sI : -sQ : sQ : sI
; Before the second inner loop(adaption operation), the real(eI)
; and imag.(eQ) components of the error term will be formatted as:
;       bits->  63...    ...0
;               eI : -eQ : -eQ : eI
; The real(hI) & imag.(hQ) terms of the filter coeff. are stored as:
;       bits->  31.. ..0
;                  hQ : hI
; The real(yI) & imag.(yQ) terms of the output are stored as:
;       bits->  31.. ..0
;                 yQ : yI
; ********************************************
_TEXT SEGMENT
; setup the pointers for sI/sQ, hI/hQ, yI/yQ.
; also set the pointers for storing the input length
; and filter coefficient length.
_sPtr$ = 28
_hPtr$ = 32
_yPtr$ = 36
_hLen$ = 40
_xLen$ = 44
PUBLIC _FilAdapAsm
_FilAdapAsm PROC NEAR USES ebx ecx edx ebp esi edi
; store sI/sQ pointer in ecx
; store hI/hQ pointer in edx
; store yI/yQ pointer in ebp
; store the input data length in di
; store the filter coeff length in loop_count
```

```
MOV         ecx, _sPtr$[esp]
MOV         edx, _hPtr$[esp]
MOV         ebp, _yPtr$[esp]
MOV         di, _xLen$[esp]
; This loop embeds two inner loops. The first inner loop performs
; the filtering operation. The second inner loop performs the adaption
; of the filter coefficients. Between the two inner loops, there is a
; scalar section of code that calculates the output and the error terms.
outerLoop:
    MOV ebx, edx                                ; copy hIQ ptr into ebx
    PXOR        mm7, mm7                        ; initialize mm7 to zeros.
                                                ; One of the partial sums in
                                                ; innerloop1 will be stored in
                                                ; mm7
    MOV si, _hLen$[esp]; copy filter length to si
                                                ; this will be the loop
                                                ; counter for innerloop1
                                                ; Note that this assumes no
                                                ; stack pushes are done
                                                ; within this code
                                                ; If needed, this can easily be
                                                ; avoided by storing this
                                                ; value in a local variable
    PXOR        mm5, mm5                        ; initialize mm5 to zeros
                                                ; mm5 will be used in
                                                ; innerloop1
                                                ; to store one of the partial
                                                ; results of complex multiply
    ADD ecx, 24                                 ; initialize ecx to point to
                                                ; first element of sIQ for
                                                ; next iteration of innerloop1
    MOV eax, ecx                                ; copy sIQ pointer to eax.
                                                ; eax will be used within
                                                ; innerloop1 to point to
                                                ; subsequent elements of sIQ
    PXOR        mm3, mm3                        ; initialize mm3 to zeros.
                                                ; One of the partial sums in
                                                ; innerloop1 will be stored in
                                                ; mm3. mm1 will also be
                                                ; used in storing
                                                ; the partial sum. But it does
                                                ; not have to be initialized to
                                                ; zeros since it is used only
                                                ; towards the end of the
                                                ; loop(after calculating the
                                                ; partial product).
; This loop performs the filtering operation. It computes the intermediate
; sum used in calculating the output terms(real and imaginary).
Inner Loop1:
    MOVD         mm1, [ebx]                     ; fetch first element
                                                ; of hIQ
    PADDD       mm7, mm3                        ; accumulate 2nd partial
                                                ; product in mm7
    MOVD        mm3, 4[ebx]                     ; fetch second element of
                                                ; hIQ
    PUNPCKLDQ   mm1, mm1                        ; copy 1st hIQ into upper
                                                ; half of mm1
```

```
    PMADDWD     mm1, [eax]                  ; complex multiply the first
                                            ; set of sIQ and Hiq
                                            ; elements
    PADDD       mm7, mm5                    ; accumulate 3rd partial
                                            ; product in mm7
    MOVD        mm5, 8[ebx]                 ; fetch third element of hIQ
    PUNPCKLDQ   mm3, mm3                    ; copy 2nd hIQ into upper
                                            ; half of mm3
    PMADDWD     mm3, 16[eax]                ; complex multiply the
                                            ; second set of sIQ and hIQ
                                            ; elements
    PUNPCKLDQ   mm5, mm5                    ; copy 3rd hIQ into upper
                                            ; half of mm5
    PMADDWD     mm5, 32[eax]                ; complex multiply the third
                                            ; set of sIQ and hIQ
                                            ; elements

    PADDD       mm7, mm1                    ; accumulate the 1st partial
                                            ; product in mm7
    ADD         eax, 48                     ; update eax to point to
                                            ; next sIQ

    ADD         ebx, 12                     ; update ebx to point to next
                                            ; hIQ
    SUB         si, 3                       ; decrement loop count by
                                            ; three
    JNZ         innerLoop1                  ; end of inner loop1
    MOVQ        mm6, const_rnd              ; fetch the rounding constant
    PADDD       mm7, mm3                    ; accumulate the partial product
                                            ; from the last iteration of innerloop1
    MOVD        mm2, pos_bias               ; fetch the positive bias for clipping
    PADDD       mm7, mm5                    ; accumulate the partial product
                                            ; from the last iteration of innerloop1
    MOVD        mm3, neg_bias               ; fetch the negative bias for clipping
    PADDD       mm7, mm6                    ; add the rounding constant to sumIQ
    PSRAD       mm7, 14                     ; signed shift right the real and imag.
                                            ; terms of sumIQ by 14 bits. The real
and
                                            ; imag. terms of the output are now in
the
                                            ; upper & lower 32 bits of mm7
                                            ; respectively
    PXOR        mm0, mm0                    ; initialize mm0 to zeros. used for
                                            ; checking the output for zero equality
    MOVQ        mm6, mm7                    ; copy the output(yIQ) into mm6
    PXOR        mm1, mm1                    ; initialize mm1 to zeros
    MOVQ        mm5, err_fmt_one            ; fetch 1st constant for formatting
                                            ; the error term

    PUNPCKHDQ   mm6, mm6                    ; copy upper half(yI) of mm6 into lower
    PUNPCKLWD   mm7, mm6                    ; pack yIQ into mm7 with bits[15..0]
                                            ; specifying the imag. term and
bits[31..16]
                                            ; specifying the real term of the
output
    MOVQ        mm4, mm7                    ; copy yIQ into mm4
```

```
    PCMPEQW     mm0, mm7                    ; check the real and imag. terms of
output
                                            ; for zero equality
    MOVD        [ebp], mm7                  ; store the output
    PCMPGTW     mm4, mm1                    ; check the real & imag. terms of
output
                                            ; for > zero
   ADD          ebp, 4                      ; update ebp to point to next output
    POR         mm0, mm4                    ; create the boolean pattern to check
if
                                            ; the output terms are >= zero
    PAND        mm2, mm0                    ; store positive bias in mm2
bits[31..0]
                                            ; if yI >= 0 and yQ >= 0
    PANDN       mm0, mm3                    ; store negative bias in mm0
bits[31..0]
                                            ; if yI < 0 and yQ < 0
    POR         mm0, mm2                    ; compute vI and vQ terms, used in
                                            ; calculating the error terms
    PXOR        mm4, mm4                    ; initialize mm4 to zeros
    MOVQ        mm1, err_fmt_two            ; fetch 2nd constant for formatting
                                            ; the error term
    PSUBW       mm0, mm7                    ; compute the diff. between vIQ and yIQ
    MOV eax, ecx                            ; copy sIQ pointer to eax. eax will
                                            ; be used within innerloop2 to
                                            ; point to subsequent elements of sIQ
    PSRAW       mm0, 4                      ; signed shift right mm0 by 4 bits
                                            ; the real & imag. terms of the error
                                            ; are now in bits [31..16] and
bits[15..0]
                                            ; respectively
     MOVQ       mm2, 16[eax]                ; fetch the 2nd sIQ element for complex
                                            ; multiplication in innerloop2. The 1st
                                            ; sIQ element will be fetched within
                                            ; innerloop2. In this case, the 2nd
                                            ; partial product is being computed
                                            ; before the first one for performance
                                            ; reasons
        PUNPCKLDQ       mm0, mm0            ; copy real & imag. error terms to
                                            ; upper half...bits[63..32]
        MOV             ebx, edx            ; copy hIQ ptr into ebx. will be used
                                            ; within innerloop2 to point to
                                            ; subsequent elements of hIQ
        PSUBW           mm4, mm0            ; compute the negative of real & imag.
                                            ; error terms in mm4..{0 - eI/eQ = -
eI/-eQ}

        PAND            mm0, mm5            ; format real(eI) & imag(eQ) error
terms in
                                            ; bits[63..48] & bits[15..0]
        PAND            mm4, mm1            ; format -eQ and -eI in bits[47..32]
                                            ; and bits[31..16] respectively
        MOVQ    mm5, const_rnd              ; fetch the rounding constant
        POR             mm4, mm0            ; get eIQ in the form: eI:-eQ:-eI:eQ

        MOV     si, _hLen$[esp]             ; copy filter length to si
                                            ; this will be the loop counter
```

```
                                                ; for the innerloop2
                                                ; Note that this assumes no stack
                                                ; pushes are done within this code
        PMADDWD mm2, mm4                        ; compute the partial product from
                                                ; complex multiplication of the 2nd
                                                ; term of sIQ and eIQ
; This loop performs the adaption operation. It calculates the new
; filter coefficient terms.
innerLoop2:

        MOVQ            mm0, [eax]              ; fetch the 1st sIQ
                                                ; element
        PADDD           mm2, mm5                ; add the rounding
                                                ; constant to
                                                ; the 2nd complex
                                                ; product
        MOVD            mm3, 4[ebx]             ; fetch the 2nd hIQ
                                                ; element
        PMADDWD         mm0, mm4                ; complex multiply
                                                ;1st sIQ with
                                                ; the error term
        MOVD            mm1, [ebx]              ; fetch the 1st hIQ element
        PSRAD           mm2, 15                 ; signed shift right 2nd partial
                                                ; product by 15 bits
        MOVQ            mm6, mm2                ; copy 2nd partial product to mm6
        ADD             eax, 16                 ; partially update eax to point to
                                                ; next sIQ.
        PADDD           mm0, mm5                ; add the rounding constant to
                                                ; the 1st complex product
        PUNPCKHDQ       mm6, mm6                ; copy the real term of the 2nd partial
                                                ; 2nd partial product in bits[31..16]
                                                ; and bits[15..0] respectively
        PUNPCKHDQ       mm7, mm7                ; copy the real term of the 1st partial
                                                ; product to lower 32 bits of mm6
        PADDSW          mm3, mm2                ; perform saturated addition of the 2nd
                                                ; partial product with hIQ
        MOVD    4[ebx], mm3                     ; store the (2nd)new filter coeffs.
        PUNPCKLWD       mm0, mm7                ; format the real & imag. terms of
                                                ; 1st partial product in bits[31..16]
                                                ; and bits[15..0] respectively
        MOVQ    mm2,    32[eax]                 ; fetch the 2nd sIQ element
        PADDSW          mm1, mm0                ; perform saturated addition of the 1st
                                                ; partial product with hIQ
        MOVD    [ebx],  mm1                     ; store the (1st)new filter coeffs.
        PMADDWD         mm2, mm4                ; complex multiply 2nd sIQ with
                                                ; the error term
        ADD             eax, 16                 ; partially update eax to point to
                                                ; next sIQ.
        ADD             ebx, 8                  ; update ebx to point to next hIQ
        SUB             si, 2                   ; decrement loop count by two
JNZ                     innerLoop2              ; end of innerLoop2

        SUB             di, 3                   ; decrement outerloop count by 3
JNZ                     outerLoop               ; end of outerloop
Done:
        ret 0
```

```
_FilAdapAsm ENDP
_TEXT   ENDS
END
```